

---

# **Sissejuhatus infotehnoloogiasse**

# Loengu ülevaade

---

- Rekursioon: üldpõhimõtted
- Imperatiivne vs deklaratiivne programmeerimine
- Funktsionaalne programmeerimine
- Loogiline programmeerimine

# Recursion

- A subroutine is said to be recursive if **it calls itself**, either directly or indirectly. That is, the subroutine **is used in its own definition**.

- Bad kind - infinite loop:

- a car is a car,

```
■ int fact(int x) {  
    return fact(x)  
}
```

- Recursion can often be used to solve complex problems by reducing them to simpler problems of the same type.

- Good kind – loop is (hopefully) terminated:

- An "ancestor" is either a parent or an ancestor of a parent.

```
■ int fact (int x) {  
    if (x <= 0) return 1;  
    else return x * fact(x-1);  
}
```

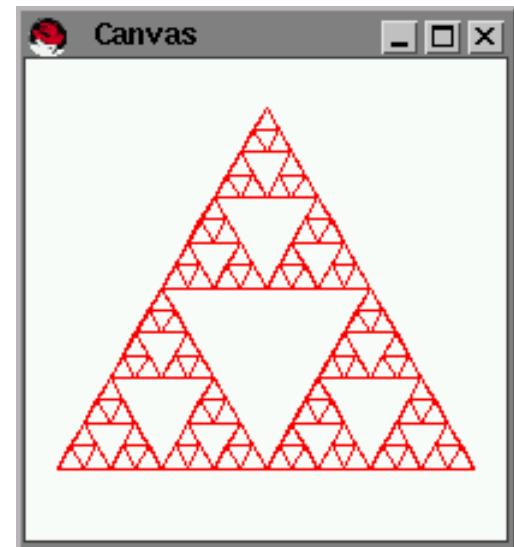
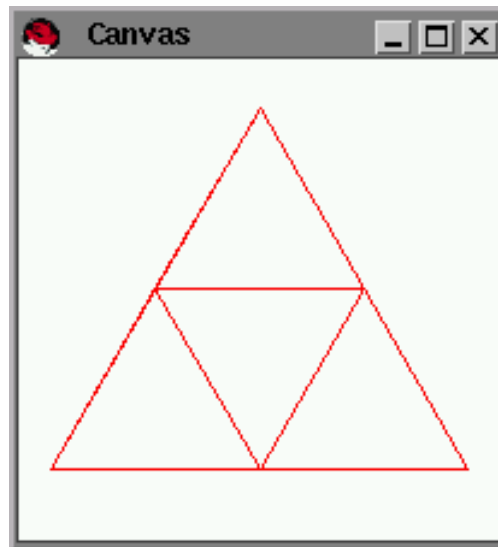
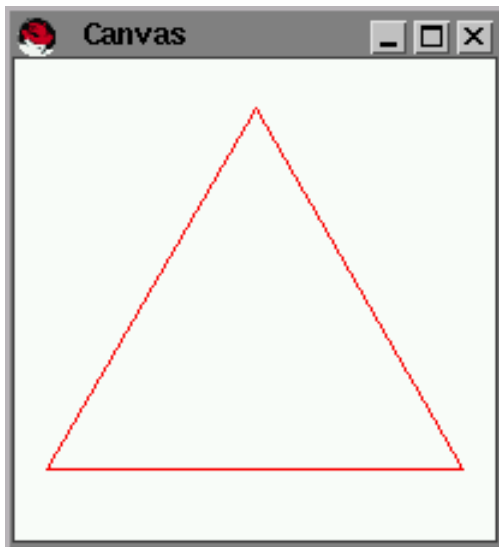
# Factorial: trace

---

```
Fact(3)
==> 3 * fact(3 - 1)
==> 3 * fact(2)
==> 3 * (2 * fact(2 - 1))
==> 3 * (2 * fact(1))
==> 3 * (2 * (1 * fact(1 - 1)))
==> 3 * (2 * (1 * fact(0)))
==> 3 * (2 * (1 * 1))
==> 6
```

# Recursive objects in the world. Fractal objects

- A huge amount of of things in the real world has a recursive nature.
- For example, coastline of a sea has a **fractal** nature – fractals are recursive: their structure is repeated over and over in small details.
- Sierpinski triangle:



# Fractals

---

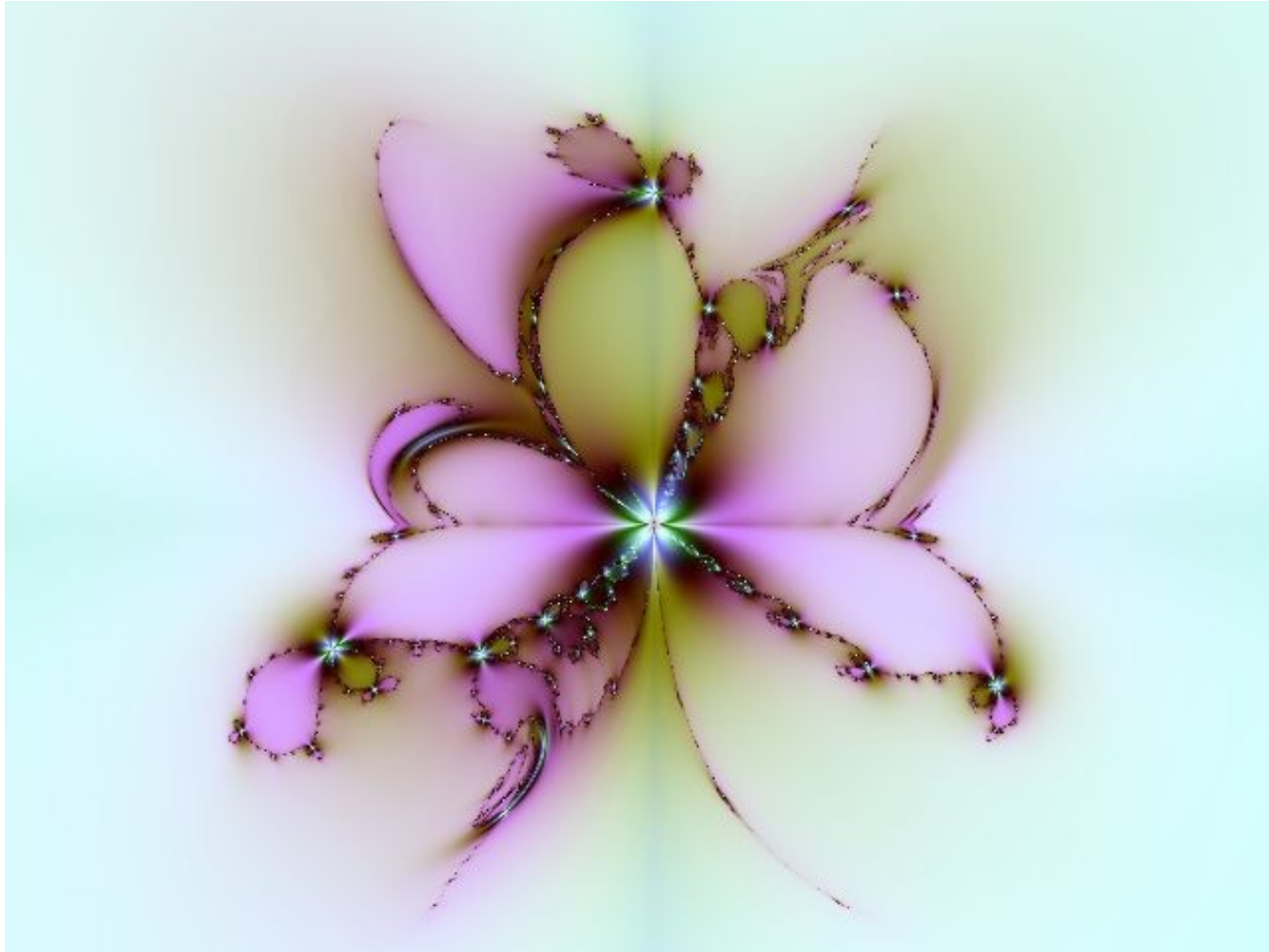
“Evening over San Francisco”



# Fractals

---

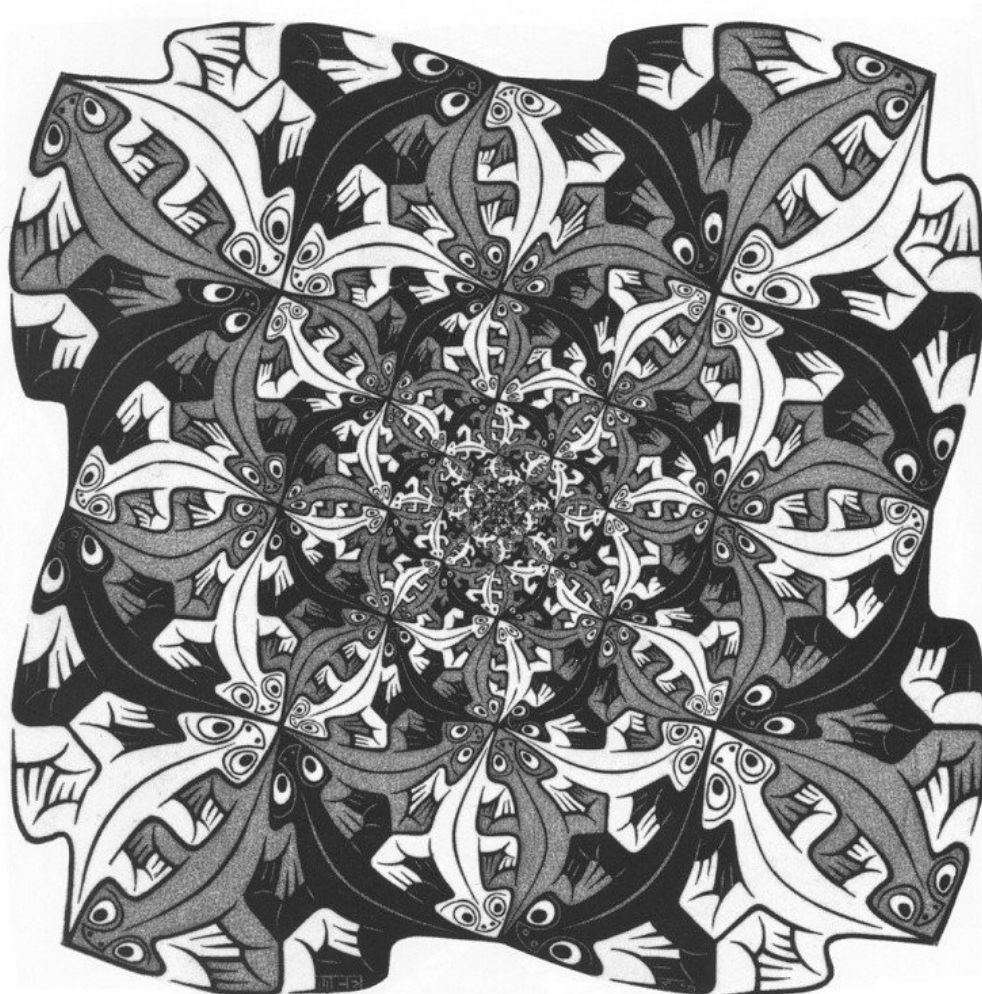
“Nature uses as little as possible”



# Fractals

---

M.C. Escher: smaller and smaller





# Fractal/recursive objects in the real world

M.C. Escher: Three worlds



# Recursive objects in the world: encoding

Recursion allows to encode a huge amount of complexity in a very efficient way.

Example:

- animals are incredibly complex
- animals are constructed from blueprints: genes
- genes are very small when compared to animals
- animals contain copies of genes in huge amounts
- How is it possible?
- Animals are constructed from gene blueprints in recursive form! Animal construction constructs many small parts similarly to large parts.

# Recursive functions: main principles

---

- Important to check that recursion terminates. Code should contain:
    - One or more **base cases** (no recursion involved!)
    - One or more **recursive cases**. Arguments of the recursive call must be “**simpler**” according to some measure.
- NB! The “simplicity” measure may be arbitrarily complex.

# Recursive functions: main principles

---

- Recursion has same power as iteration:
  - Every recursive function can be written using while or for loops instead
  - Every function using while and/or for loops can be written using recursion instead
- However:
  - some programming tasks are much easier to write using recursion
  - some programming tasks are much easier to write using iteration

# Direct and indirect recursive call

- A recursive subroutine is one that calls itself, either directly or indirectly.
- To say that a subroutine calls itself **directly** means that its definition contains a subroutine call statement that calls the subroutine that is being defined.
  - foo calls foo:
  - `int foo(int x) { if (x>0) return 1+foo(x-1) else return 1 }`
- To say that a subroutine calls itself **indirectly** means that it calls a second subroutine which in turn calls the first subroutine (either directly or indirectly).
  - foo calls bar which calls foo:
  - `int foo(int x) { if (x>0) return 2+bar(x-2) else return 1 }`
  - `int bar(int x) { if (x>0) return 2*+foo(x-1) else return 1 }`

# Tail recursion: directly convertible to iteration

- Tail recursion: recursive call is the last thing the function does.
- Tail recursive functions are typically much faster when written iteratively (unless we have a very good optimising compiler)
- **Tail recursive:**
  - `int foo(int x) { if (x>0) return foo(x-2) else return 1 }`
  - `int foo(int x) { if (x>0) return foo(x-1)  
else if (x<0) return 1 else return foo(x-2) }`
- **Not tail recursive:**
  - `int foo(int x) { if (x>0) return foo(x-2)+foo(x-1) else return 1 }`
  - `int foo(int x) { if (x>0) return 1+foo(x-1) else return 1 }`

# Graphical examples

---

- See, modify and program graphical examples from Eck xTurtle lab 3:

- Simple examples:

<http://sise.ttu.ee/it/vorgutarkvara/wai2030/eckintrolabs/tmcm-java-labs/labs/xTurtleLab3.html>

- A bit more complex examples:

<http://math.hws.edu/TMCM/java/xTurtle/index.html>

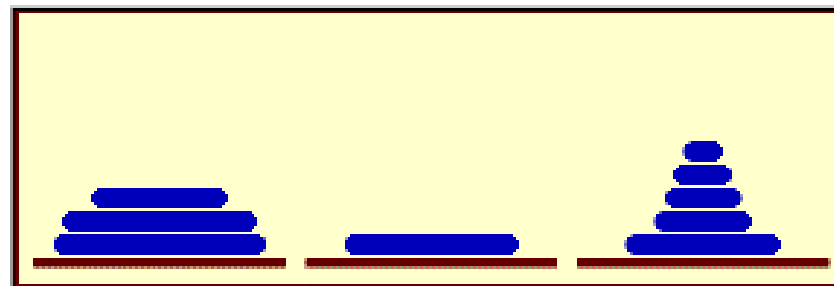
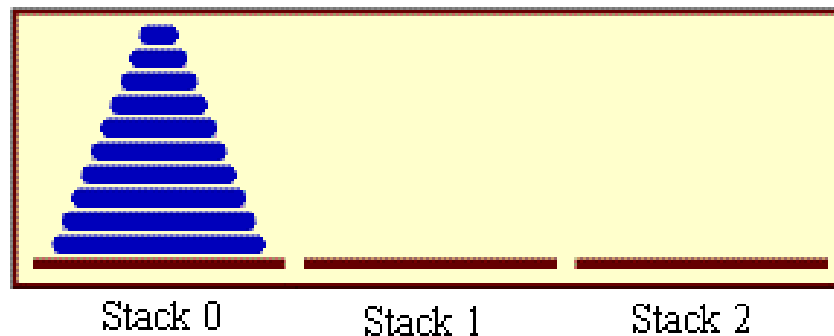
# Recursive binary search

```
static int binarySearch(int[] A, int loIndex, int hiIndex, int value) {  
  
    if (loIndex > hiIndex) {  
        return -1;  
    }  
    else {  
        int middle = (loIndex + hiIndex) / 2;  
        if (value == A[middle])  
            return middle;  
        else if (value < A[middle])  
            return binarySearch(A, loIndex, middle - 1, value);  
        else // value must be > A[middle]  
            return binarySearch(A, middle + 1, hiIndex, value);  
    }  
} // end binarySearch()
```



# Example: towers of Hanoi

- Goal: move all the disks from Stack 0 to Stack 1
- Stack 2 can be used as a spare location.
- Only one disk at a time may be moved
- A larger disk may never be on the smaller disk



The stacks after a number of moves.

# Towers of Hanoi: code

---

```
void TowersOfHanoi(int disks, int from, int to, int spare) {  
    if (disks == 1) {  
        System.out.println("Move a disk from stack number "  
            + from + " to stack number " + to);  
    } else {  
        TowersOfHanoi(disks-1, from, spare, to);  
        System.out.println("Move a disk from stack number "  
            + from + " to stack number " + to);  
        TowersOfHanoi(disks-1, spare, to, from);  
    }  
}
```

# Deklaratiivne vs imperatiivne

---

- Programmeerimiskeeli ja -meetodeid saab klassifitseerida mitmel moel. Selle loengu kontekstis sobib jaotada programmeerimiskeeled kõigepealt kahte gruppi:
- Imperatiivsed keeled] sobivad samm-sammult, kindlas järjekorras täidetavate algoritmide esitamiseks. Programmid kujutavad endast arvutile antavate käskude jada. Tuntumad imperatiivsed keeled on C, Basic, Pascal, Java, objektorienteeritud keeled ja assemblerkeeled.
- Imperatiivsete keelte peamiseks **eeliseks** on arvuti tegevuse täpse kontrollimise ja suunamise võimaldamine, mis enamasti tagab maksimaalse töökiiruse.
- **Miinusteks** on programmeerimise suur töömahukus - lahenduskäigu kõik detailid tuleb süsteemile esitada - ning suured raskused programmideanalüüsimisel, näiteks optimeerimise, verifitseerimise või paralleliseerimise tarvis.

# Deklaratiivsed vs imperatiivsed keeled

---

- **Deklaratiivsed keeled** sobivad algoritmi esitamiseks käskude jadast abstraktsemal viisil. Programmeerija ei pruugi alati kõiki algoritmi detaile kirja panna, vaid võib esitada otsitava lahenduse kirjelduse, ning juba programmi täitmise käigus otsustab süsteem automaatselt, mis viisil täpselt seda lahendust otsida.
- Deklaratiivseteks keelteks võib lugeda loogilise programmeerimise keeled (näiteks Prolog) ja mitmed funktsionaalsed keeled (näiteks Haskell). Teoorias kasutatav lambda-arvutus on puhtalt funktsionaalse deklaratiiivse keele näide.

# Plussid ja miinused

---

- Deklaratiivsed keeled võimaldavad enamikku programme kiiremini ja mugavamalt kirjutada, kui imperatiivsed keeled - programmeerija ei pea kõigi detailide eest hoolt kandma. Tunduvalt lihtsam on ka programmide analüüs, näiteks programmi automaatsel kohandamisel paralleelarvutile, kus programmi täitmise juures töötab samaaegselt hulk protsessoreid.
- Peamiseks miinuseks on programmide väiksem töökiirus - deklaratiivne programm ei pruugi küll alati aeglasem olla, kui imperatiivne, kuid on seda harilikult siiski. Põhjuseks on siin keele automaattranslaatori väiksem intelligentsus kogenud programmeerijaga võrreldes.

# Funktsionaalsed ja loogilised keeled

---

- Deklaratiivsed keeled jaotatakse
  - **Funktsionaalse programmeerimise keelteks** (näide: Haskell), kus lahendus kirjeldatakse funktsioonide kogu abil - ka viimast saab tegelikult käsitleda kui teatud tüüpi loogikasüsteemi.
  - **Loogilise programmeerimise keelteks** (näide: Prolog), kus otsitavat lahendust kirjeldatakse loogika keeles

# Funktsionaalne programmeerimine

---

- Funktsionaalsete keelte idee on programmide kirjutamine (matemaatiliste) funktsioonide defineerimise teel, määramata seejuures täpselt ära, mis strateegia järgi funktsiooni resultaati tuleb arvutada.
- Funktsioonile võib anda argumendiks teisi funktsioone ja funktsiooni arvutamise resultaadiks võib samuti olla funktsioon.
- NB! Funktsionaalne programmeerimine on ajalooliselt esimene **deklaratiivse programmeerimise** viis, ning enamik kaasaegseid programmeerimiskeeli -- C, Pascal, Ada jne -- on funktsionaalse programmeerimise meetoditest mõjustatud.

# Näide: faktoriaal

- Defineerime näiteks faktoriaali funktsiooni `fakt` ja funktsiooni `map`, mis rakendab oma esimest, funktsionaalset argumenti teiseks argumendiks oleva loendi igale elemendile.
- Loend `[h|t]` esitatakse sisemiselt kui harilik term `. (h, t)`

```
fakt(x, 0) = 1
```

```
fakt(x, n) = x*fakt(x-1)
```

```
(ehk: fakt(x) = if x=0 then 1 else x*fakt(x-1))
```

```
map(f, []) = []
```

```
map(f, [h|t]) = [f(h) | map(f, t)]
```

- Avaldise `map(fakt, [3, 5, 0])` väärtuseks on `[6, 120, 1]`.



# Avaldise väärtustamine (reduktsioon)

- Leitakse funktsiooni defineeriv "sobiv" vördus
- Asendatakse avaldis vörduse parema poolega
- Formaalseste parameetrid asendatakse tegelike argumentidega
- Kogu protsessi korratakse kuni rohkem ei saa

```
fact 3
==> 3 * fact (3 - 1)
==> 3 * fact 2
==> 3 * (2 * fact (2 - 1))
==> 3 * (2 * fact 1)
==> 3 * (2 * (1 * fact (1 - 1)))
==> 3 * (2 * (1 * fact 0))
==> 3 * (2 * (1 * 1))
==> 6
```

# Puhtad ja ebapuhtad keeled

---

- Funktsionaalseid keeli saab jämedalt jagada kahte liiki: puhtad ja ebapuhtad.
- Puhtas keeles -- Haskell, Hope, Miranda, FP -- ei ole programmeerijal peale funktsioonide defineerimise ja sisseehitatud baasfunktsioonide (aritmeetika, loendid jms) mingeid lisavahendeid -- kõik kõrvalefektid on keelatud.
- Puhas funktsionaalne keel ei luba muutujatele väärtusi omistada. Ainus efekt, mis funktsiooni rakendamine argumentidele annab, on resultaadi leidmine.
- Ebapuhtad funktsionaalsed keeled - ML, Lisp, Scheme - kombineerivad puhaste funktsionaalsete keelte mehhanisme imperatiivsete mehhanismidega.

# Alus: lambda-arvutus

---

- Lambda-arvutuse keel on Alonzo Churchi poolt 1930. aastatel leiutatud lihtne ja universaalne meetod funktsioonide kirjapanekuks.
- Lambda-arvutuse teooria tegeleb arvutatavuse ja arvutatavate funktsioonide uurimisega, kasutades selleks lambda-arvutuse keelt kui universaalset programmeerimiskeelt.
- Churchi tees väidab, et iga algoritmi saab lambda-arvutuse keeles kirja panna. On võimalik näidata, et lambda-arvutus, nagu ka Prolog, C ja Basic on üks paljudest universaalsetest programmeerimiskeeltest.
- Konkreetselt on lambda-arvutuse keel ja teooria funktsionaalsete programmeerimiskeelte aluseks.

# Annonüümsed funktsioonid

- üks harilikumaid praktikas kasutatavaid funktsioonide kirjapaneku viise on selline:

$$f(x) = x * x + 1$$

- Funktsioon esitatakse, andes talle samas nime, konkreetse näites  $f$ . Lambda-arvutuses esitatakse funktsioone, vastupidi, kui anonüümseid, nimeta terme. äsjatoodud näide on lambda-kirjaviisis

$$\lambda x. x * x + 1$$

- Lambda-sümboli  $\lambda$  järele kirjutatakse funktsiooni formaalseks parameetriks olev muutuja, seejärel punkt ja funktsiooni keha.

# Lambda-arvutus

---

- Mitme formaalse parameetriga funktsioone esitatakse mitme üksteise sees oleva üheparameetrilise funktsioonina:

$\lambda x. \lambda y. x*x+y*y.$

# Näide

- Funktsiooni rakendamiseks kirjutatakse traditsioonilise  $f(3)$  asemel  $(\lambda x. x*x+1) 3$  -- viimase väärtuseks on 10.
- Analoogiliselt annab  $((\lambda x. \lambda y. x*x+y*y) 2) 3$  väärtuseks 13. Lambda-termi rakendamisel asendatakse seotud muutuja termi kehas termile antud argumendiga.

$((\lambda f. f(f 2)) (\lambda x. x*x+1))$  annab  
 $(\lambda x. x*x+1) ((\lambda x. x*x+1) 2)$  annab  
 $(\lambda x. x*x+1) (2*2+1)$  annab  
 $(\lambda x. x*x+1) 5$  annab  
 $5*5+1$  annab  
26.

Resultaat seejuures asenduste tegemise järjekorrast ei sõltu.

# Ebapuhta keele näide: Scheme

---

```
(define (good? x)
  (if (> 10 x) #t #f) )
```

```
(define (every? fn lst)
  (if (pair? lst)
      (if (fn (car lst))
          (every? fn (cdr lst))
          #f)
      #t))
```

```
(every? good? `(10 40 50 2 100))
```

# Loogiline programmeerimine: Prolog

---

- Prolog on esimene -- ja siiani väga populaarne -- loogilise programmeerimise keel. Prolog-ile lisaks on välja töötatud mitmeid uuemaid loogilise programmeerimise keeli ja süsteeme, ning nende arendamine on ulatuslik ja levinud uurimisteema.
- Prolog-i põhi-idee on nõuda otsitava lahenduse kirjeldamist esimest järku predikaatarvutuse keeles, kusjuures Prolog-i süsteem sisaldab teatud tüüpi automaatset teoreemitõetajat, mis on võimeline lahendust automaatselt otsima ja tuletama.
- Sellegipoolest ei ole Prolog siiski automaatse teoreemitõestamise süsteem: viimast realiseeriv mehhanism on Prolog-is väga piiratud, spetsiifiline ja loogiliselt mittetäielik.



- Vaatleme esimese näitena sugulussidemete andmebaasi, konkreetselt niisugust Prolog-i programmi:

```
isa(jaan,peeter).  
isa(jaan,martin).  
isa(martin,veiko).  
isa(riivo,leo).  
ema(leena,leo).  
vanaisa(X,Z) :- isa(X,Y), isa(Y,Z).  
vanaisa(X,Z) :- isa(X,Y), ema(Y,Z).
```

# Päringud

- Päringule, kas `isa(riivo,martin)` on antud andmebaasist tuletatav, vastab otsingumootor eitavalt

```
?- isa(riivo,martin).
```

```
no
```

- Lahendust otsib Prologi mootor järgmiselt: kõik andmebaasis olevad laused vaadatakse järjest läbi ning püütakse iga lause esimest literaali päringu-literaaliga unifitseerida. Kui see ei õnnestu, vastatakse päringule eitavalt.

- Päringule, kas `isa(riivo,leo)` on antud andmebaasist tuletatav, vastab otsingumootor jaatavalt:

```
?- isa(riivo,leo).
```

```
yes
```

# Päringud

- Päringud võivad sisaldada muutujaid: sellisel juhul leiab Prologi mootor lahenduse, st. muutuja asemele substitueeritava termi:

```
?- isa(jaan,X) .
```

```
X=peeter
```

- Kui me ei ole rahul esimese leitud lahendusega, võime otsingumootorit instrueerida uusi lahendusi otsima, st püüdma unifitseerida päringut järgmiste andmebaasis olevate lausetega.

```
?- isa(jaan,X) .
```

```
X=peeter;
```

```
X=martin;
```

```
no
```

# Reeglite kasutamine

---

- kuidas reageerib otsingumootor päringule, kas Jaan on kellegi vanaisa:

```
?- vanaisa(jaan,X) .
```

```
X=veiko
```

# Loendi element?

- Kirjutame programmi `member` , mis kontrollib, kas esimene argument kuulub teiseks argumendiks olevasse loendisse:

```
member(X, [X|Z]) .
```

```
member(X, [Y|Z]) :- member(X, Z) .
```

```
?- member(10, [20, 30, 10, 40]) .
```

Yes

# Liida loendid

```
append([ ], Z, Z) .
```

```
append([X|Y], Z, [X|R]) :- append(Y, Z, R) .
```

```
?- append([a,b], [c,d], L) .
```

```
L=[a,b,c,d];
```

```
no
```

```
?- append(X, Y, [a,b,c,d]) .
```

```
X=[ ], Y=[a,b,c,d];
```

```
X=[a], Y=[b,c,d];
```

```
X=[a,b], Y=[c,d];
```

```
X=[a,b,c], Y=[d];
```

```
X=[a,b,c,d], Y=[ ];
```

```
no
```

# Permutatsioonid

```
perm(L, [H|T]) :- app(V, [H|U], L), app(V, U, W),  
                  perm(W, T).  
perm([], []).
```

```
?- perm([a,b,c], X).  
X=[a,b,c];  
X=[a,c,b];  
X=[b,a,c];  
X=[b,c,a];  
X=[c,a,b];  
X=[c,b,a];  
no
```